[0048] Instruction 1 fetches a unique internal task identifier from the current thread data structure (whose pointer is permanently allocated to a register). This task identifier is encoded as an offset from the beginning of a task class mirror table to the initialized entry that is assigned to that task. Instruction 2 fetches the address of the task class mirror from the task class mirror table. A null pointer at this location indicates that the class has not yet been initialized by the current task.

[0049] Instruction 3 branches to the label "end_barrier" if there is a non-null pointer in the task class mirror table, therefore bypassing the call to the initialization code. Instruction 4 is executed in the delay slot of the branch at instruction 3 only if the branch is taken (delay slot is annulled otherwise). Instruction 4 loads the static variable from the task class mirror object (the offset to the static variable has typically been set before the class barrier in a register, here the static_var_offset register).

[0050] If the pointer loaded by instruction 2 is null, the branch at instruction 3 is not taken and execution falls through to instruction 5.

[0051] Note that instruction 3 is a branch on register value, a common instruction in RISC processors. This type of branch instruction is favored in the implementation of the barrier over more conventional branch on condition instructions since it eliminates the need for an extra comparison instruction. Instruction 5 calls a stub to a runtime function that initializes the class for the task executing this barrier. The stub takes care of setting the register "tcm" to the task class mirror of the initialized class upon returns, so that instruction 7 can load the static variable. After the end_barrier label, the tmp register holds the static variable value.

[0052] Instructions 4 and 7 can be replaced with a store instruction if the barrier sequence implements an assignment to a static variable, i.e.:
st tmp, [tcm + static_var_offset], where the register tmp holds the value assigned to the static variable.

[0053] If the barrier is not for a static variable access, then instruction 4 can be replaced with a "no operation" (nop) instruction and instruction 7 can be removed, or both 4 and 7 can be replace with a useful instruction.

[0054] The invention also provides with fast access to the task class mirror of a class for a given task when class initialization tests can be omitted (for instance, access to a variable of a class from its class initialization code can avoid a class initialization barrier while leaving the initialized entry set to null). In this case, access to the task class mirror is provided by the resolved entry associated with the task in the task class mirror table. This entry is set to the task class mirror at class load time. The following code illustrates how access to the class time mirror is provided in this case:

```
1       ld [gthread + encoded_task_id], initialized_entry_offset
2       add tcm_table, TCM_POINTER_SIZE, tcm_table
3       ld [tcm_table +initialized_entry_offset], tcm
4       ld [tcm + static_var_offset]
```

[0055] As before, instruction 1 fetches the current task's unique internal task identifier encoded as an offset to the initialized entry assigned to that task. Instruction 2 adds the size of a task class mirror pointer to the pointer to the task class mirror table so that adding the initialized entry's offset to it will actually give the address of the resolved entry (since this one is located immediately after the initialized one). This strategy avoids storing another encoded task id in the thread descriptor if space consumption is a concern. Alternatively, it is possible to eliminate instruction 2 by storing also in the thread descriptor a second encoded task identifier whose value is an offset to the resolved entry assigned to the task. Instruction 3 fetches the task class mirror, and instruction 4 loads the desired class variable.

[0056] The class initialization barrier and static variable access mechanisms described above ([0047] to [0055]) can be simply modified to also take into account multitasking virtual machine implementations that treat initialization-less classes specially in order to minimize the space consumed by task class mirror tables, as described earlier in [0015]. With this approach, an initialization-less class is associated with a task class mirror table that includes only one entry per task. This entry is set upon loading of the initialization-less class by the corresponding task. Classes that require initialization are, as described before, associated with a task class mirror table that includes two entries per task. However, as noted earlier ([0015]), the entries in the class must be arranged such that all the initialized entries are placed contiguously in the table, followed by all the resolved entries. This arrangement enables the encoding of a task identifier into an offset to an initialized entry of a task table to be used without change for both initialization-less classes and classes that require initialization. A dynamic compiler does not generate any class initialization barrier for initialization-less classes, but must nevertheless generate the level of indirection necessary to access a task's copy of the variables of a class. The code generated in this case consists of only Instructions 1, 2 and 7 of the sequence of instructions described in [0047]. It may not always be possible to eliminate the class barrier for initialization-less classes (e.g., the implementation of an interpreter may only use the generic version of static variable access in order to reduce the number of platform-independent instructions used at runtime), in